THE UNIVERSITY OF WAIKATO
Department of Computer Science

COMP204-24B — Practical Networking & Cyber Security

A3 - TFTP

1 Introduction

In this assignment, you will implement a client/server system to reliably transfer files using UDP sockets. You will implement a variant of the Trivial File Transfer Protocol (TFTP), which is an open Internet standard published in RFC 1350. The protocol you implement will be slightly simplified from the TFTP protocol. As a guideline, around 200 lines of source Java code are sufficient to implement the ability to send a file from the server, and about the same number for the client.

This assignment will introduce you using the DatagramSocket and DatagramPacket Java classes. Please take some time to look through the Java documentation provided for these classes. You can find the documentation at

`https://devdocs.io/openjdk~8/`

2 Academic Integrity

The files you submit **must** be your own work. You may discuss the assignment with others but the actual code you submit must be your own. You must fully also understand your code and be capable of reproducing and modifying it. If there is anything in your code that you don't understand, seek help until you do.

You **must** submit your files to Moodle in order to receive any marks recorded on your verification page.

You **must** demonstrate your code in the lab to one of the teaching staff within one week of the deadline to have the marks counted.

This assignment is worth 10% of your final grade.

3 TFTP

TFTP servers usually listen on port 69, using the User Datagram Protocol (UDP). TFTP servers are widely used as the first stage in network booting to fetch the operating system kernel for a system to boot. In this case, the client opens a UDP socket and sends a read request (RRQ) with the name of the file to fetch. If the file does not exist on the server, the server sends the client an

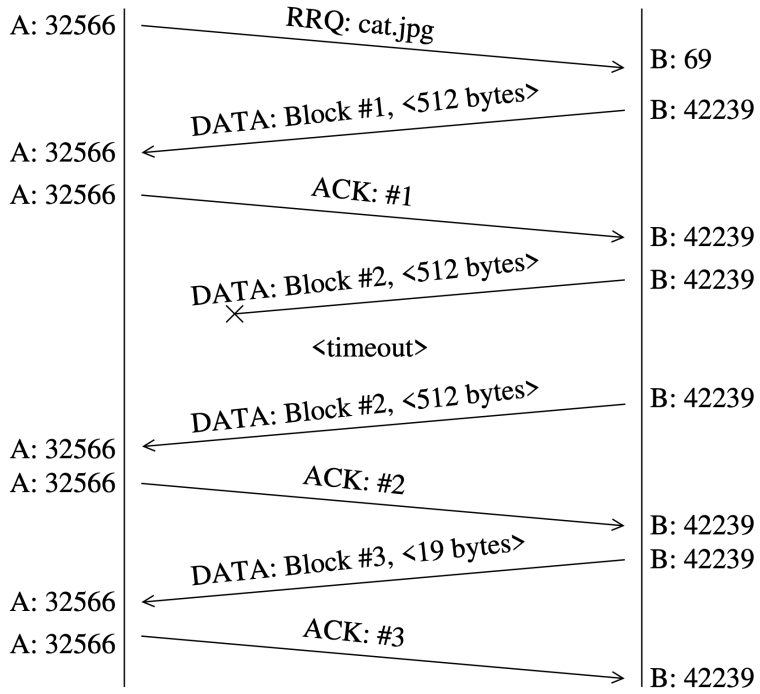


Figure 1: Time diagram illustrating the communication in TFTP.

ERROR packet. Otherwise, the server opens a new DatagramSocket bound to a new port, and then sends one DATA packet at a time to the client, each numbered to distinguish this packet from the last. The server waits for the client to ACK the received packet, before sending the next DATA packet. If the server does not receive an ACK from the client after a timeout, the server will re-transmit the DATA packet using the same block number.

The server sends DATA packets containing 512 bytes of data at a time. The client infers that end of file (EOF) has been reached when it receives a DATA packet containing less than 512 bytes of data. If the file is a multiple of 512 bytes in size, the server signals EOF with a packet with zero bytes of data.

The figure shows an example transfer of cat.jpg of 1043 bytes in size from server B to client A. A sends the request to port 69 on B; B allocates a new DatagramSocket on an unused port (42239) and sends it back to A who is listening on port 32566 for the DATA. The second block is transferred twice, as the second block is lost on the path to the client. B re-transmits the block until it receives an ACK for it. Note that the ACK could have been lost instead of the DATA block; if this happens, the client must re-send the ACK, but skip over the data that it has already written to disk. For this assignment, the server should give up if it transmits the same block five times and receives no acknowledgment. The third block contains less than 512 bytes and signals end of transmission.

3.1 Skeleton code

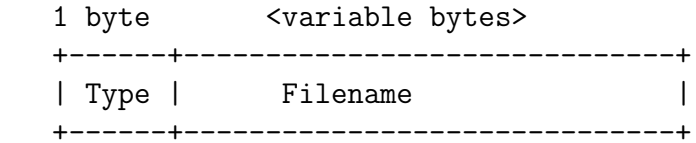
We provide the basic skeleton of a TftpServer implementation to get you started. You do not have to use it. We have left comments that describe roughly the process you should use. Briefly, the simplest implementation strategy is where the server responds to each TftpClient request using a different thread. You need to implement a retransmission method, in case a data or acknowledgment packet is lost, and you should use the DatagramSocket::setSoTimeout method for this.

3.2 Packet Formats

Every packet begins with a 1 byte type field. The defined types are:

- RRQ: 1
- DATA: 2
- ACK: 3
- ERROR: 4

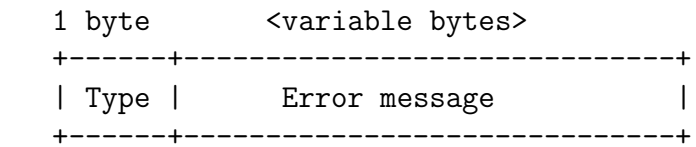
To start with, implement the "Read Request" (RRQ) exchange in both the client and the server. Have the server listen on a port it can bind to (not 69). The RRQ packet contains a String of the requested filename after the type field:



Format of RRQ packet.

To obtain a byte array from a String, use the getBytes() method on the String. To generate a String from a byte array, use the appropriate String constructor. Use the DatagramPacket::getLength field to determine the size of the packet, and thus be able to compute the size of the filename, in bytes.

If the file does not exist on the server, send an error packet back. The ERROR packet contains a String that could explain why the request could not be fulfilled, such as because the file does not exist. Print the error message in the client.



Format of ERROR packet.

If the file does exist, create a new thread in the server which will send the DATA back using a new DatagramSocket. Send the first block using block number #1. Put up to 512 bytes of data in each packet: a full size DatagramPacket, including the type and block number, contains 514 bytes of data. Send the DatagramPackets to the IP and port number the client made the request from.

```
1 byte 1 byte        <variable bytes>
+------+-------+-------------------------+
| Type | Block |      Data               |
+------+-------+-------------------------+
Format of DATA packet.
```

The client sends an ACK packet when it receives a DATA packet it expects: the block number has to be either:

1. the block it is expecting, or
2. the block it received last time.

The second condition occurs if the ACK is lost in the network being carried back to the server. The block number in the ACK packet is the in-order packet your client last received.

```
1 byte 1 byte
+------+-------+
| Type | Block |
+------+-------+
Format of ACK packet.
```

Note that the block number will wrap to zero as the maximum value that can go into a byte is 255. To test your implementation, transfer a file at least 131,072 bytes in size. Verify that the received file is correct using the md5sum command to compare the checksums of the sent and received files.

In the server, use setSoTimeout on the DatagramSocket object to set a 1 second timeout before calling the recv method to wait for an ACK. The DatagramSocket will throw an SocketTimeoutException if it does not receive an ACK in a second.

3.3 Interoperability

You should test your client and server implementation with that of a classmate. Ensure they behave correctly. If they do not, work with your classmate to determine whose implementation (or both) has a problem according to the specification provided.